What Scientific Applications Can Benefit from Hardware Transactional Memory?

Martin Schindewolf*†, Barna Bihari†, John Gyllenhaal†, Martin Schulz†, Amy Wang‡, Wolfgang Karl*

*Karlsruhe Institute of Technology, {schindewolf,karl}@kit.edu

†Lawrence Livermore National Laboratory, {bihari1,gyllenhaal1,schulzm}@llnl.gov

‡IBM/Toronto, aktwang@ca.ibm.com

***Abstract*—Achieving efficient and correct synchronization of multiple threads is a difficult and error-prone task at small scale and, as we march towards extreme scale computing, will be even more challenging when the resulting application is supposed to utilize millions of cores efficiently. Transactional Memory (TM) is a promising technique to ease the burden on the programmer, but only recently has become available on commercial hardware in the new Blue Gene/Q system and hence the real benefit for realistic applications has not been studied yet.**

This paper presents the first performance results of TM embedded into OpenMP on a prototype system of BG/Q and characterizes code properties that will likely lead to benefits when augmented with TM primitives. We first study the influence of thread count, environment variables and memory layout on TM performance and identify code properties that will yield performance gains with TM. Second, we evaluate the combination of OpenMP with multiple synchronization primitives on top of MPI to determine suitable task to thread ratios per node. Finally, we condense our findings into a set of best practices. These are applied to a Monte Carlo Benchmark and a Smoothed Particle Hydrodynamics method. In both cases an optimized TM version, executed with 64 threads on one node, outperforms a simple TM implementation. MCB with optimized TM yields a speedup of 27.45 over baseline.

I. Introduction

Achieving efficient and correct synchronization of multiple threads is a difficult and error-prone task. The correct use of lock-based schemes requires a strict coding discipline to place matching lock and unlock operations into the code in a way that avoids race conditions and/or deadlocks. Additionally, lock-based synchronization often leads to high-overheads, either due to lock contention, when using coarse-grained locks, or unnecessary lock overhead, when using fine-grained locks. This not only slows down the process using the locks, but also has a global effect in large scale programming since it creates skew between processes as well as load imbalance, both major factors limiting the scalability of applications.

Transactional Memory (TM) has been proposed almost two decades ago to tackle these issues in shared memory systems [1]. TM simplifies synchronization by providing a single simple construct: the programmer wraps the critical instructions in a transaction (also called atomic block). These transactions are then executed optimistically in parallel and conflicting accesses are resolved by a TM run time system. As a consequence only the effects of entire and completed transactions are visible to concurrent threads, avoiding the visibility of intermediate memory states.

Except for a few, by now discontinued prototype implementations in research processors, TM has mainly been confined to software solutions and therefore has been burdened with significant runtime overheads. These overheads severely restrict its applicability with the consequence that non performance critical areas, for which the increase in programmability and ease of verification justify the additional cost, are the primary target. In high performance computing, however, the applicability of these approaches has been limited.

The recently introduced Blue Gene/Q (BG/Q) system by IBM for the first time provides Hardware Transactional Memory (HTM) in a commercially available platform [2]. BG/Q is designed as a large scale platform for scientific computing workloads. The first machine is installed at Lawrence Livermore National Laboratory and provides more than 1.5 million compute cores / 6 million hardware threads, making scalability one of the premier challenges on this machine.

This paper presents the first comprehensive performance evaluation of BGQ's HTM capabilities from the application's perspective, although other recent papers have also measured certain aspects of performance [3] and [4]. Not every lock-based application will be suitable for HTM and it is important to understand what code properties lead to efficient executions and, hence, which codes can benefit from using HTM. In order to help code developers with this task, we provide a precise evaluation of the strengths and weaknesses of the architecture as well as what is required to map applications to the architecture in an efficient way. In particular, we focus on the synchronization primitives for parallel programming in shared memory architectures with OpenMP. Our experiments take into account the application's characteristic (high or low contention), the influence of environment variables, the effects of enlarging transaction sizes, and hybrid parallelization with MPI. We apply our results to the optimization of a Monte Carlo Benchmark (MCB), which functions as a proxy application for several large scale Monte Carlo simulations, and a Smoothed Particle Hydrodynamics method from the PARSEC benchmark [5].

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

Specifically, we make the following contributions:

1) We introduce a new benchmark, CLOMP-TM, that is aimed at evaluating TM systems for scientific workloads.
2) We characterize the performance of HTM combined with OpenMP on BG/Q using CLOMP-TM.
3) We study the influence of thread count, environment variables and memory layout on TM performance.
4) We determine a fitting task to thread ratio for hybrid MPI/OpenMP codes with different synchronization primitives.
5) We identify code properties that are likely to yield performance gains with TM.
6) We condense the findings into best practices and apply them to a realistic Monte Carlo Benchmark code and a Smoothed Particle Hydrodynamics method.

For both case studies, an optimized TM version, executed with 64 threads on one node, significantly outperforms a simple or naive TM version validating the best practices derived from our observations with CLOMP-TM.

The remainder of this paper is organized as follows. Section II provides background on Transactional Memory in general as well as related work. Section III describes our experimental setup, the TM architecture of the BG/Q system, and our benchmark used to determine overheads. Section IV presents low-level measurements, followed by the lessons learned in Section V. Section VI shows how we can use our lessons to add transactions to a Monte Carlo code and to a Smoothed Particle Hydrodynamics method. Section VII concludes and presents ideas for future work.

II. Background and Related Work

Transactional Memory has been proposed as architectural support for lock-free data structures in shared memory systems [1]. The core idea is to replace pessimistic synchronization, such as locks, with optimistic synchronization in the form of transactions. Programmers can group updates into transactions and these can be executed concurrently with the rest of the program. A runtime system (in hardware or software) detects conflicts between transactions as well as between a transaction and the rest of the program and, if necessary, aborts and rolls the effects of the transaction back. As a consequence, the effects of any transaction are seen as if the transaction occurred as one atomic block, providing the necessary synchronization guarantees.

This concept does not only have performance implications, since contention free cases have the potential to execute concurrently and are not forced to be serialized as with pessimistic schemes, but it also, and perhaps more importantly, has the potential to provide a boost in programmability and maintainability. Lock-based schemes are often error prone and require a strict coding discipline to ensure proper synchronization by placing matching lock and unlock instructions without causing deadlocks. The latter is particularly critical for fine grain locks. TM approaches, on the other hand, only expose a single concept, a transaction, to the user in the form of code blocks, which can be easily annotated within the source code.

Many different TM designs have been proposed [6]. Software-based approaches [7], [8], [9], [10] use a Software Transactional Memory (STM) library to implement algorithms for the detection and resolution of conflicting memory accesses. Software is very flexible but also comes with inherent overheads [11]. Hardware Transactional Memory (HTM) systems, on the other side, are fast for transactions that fit into the restricted hardware [12], [13], [14]. Further, hybrid approaches, combining hardware and software to accelerate execution are a topic of current research [15], [16], [17], [18].

The only paper describing an early experience with a commercial hardware transactional memory implementation published in a major conference, to our knowledge, is by Dice et al. [19]. The paper describes and evaluates the hardware transactional memory feature of SUN's Rock processor [20], which is no longer available, and focuses on the evaluation of concurrent data structures such as Red Black trees and `Hashtable`, and the construction of a minimum spanning forest [21]. The parallelization of these codes uses threads only. Thus, no experiments are made that estimate the performance of a hybrid parallelization with MPI. Further, there is an important difference between the HTM implementations of Rock and BG/Q. Rock is a checkpoint-based architecture which is exploited in the context of TM to save and restore the architectural state of the registers in hardware. In BG/Q the TM runtime performs this task in software, while only conflict detection is done in hardware. This important difference will affect the performance of both architectures and makes transferring results of previous studies from the Rock to the BG/Q architecture extremely difficult.

A description of a second commercial HTM implementation can be found in a paper by Click [22]. The goal is to accelerate the `synchronized` keyword in Java. Thus, no extensions to the language are made and explicit programming with transactions is not possible. Instead a heuristic decides whether to run a critical section as a transaction or not.

Most STM papers use STAMP, a benchmark suite for transactional memory research [23]. The codes comprise: Bayesian network learning, gene sequencing, network intrusion detection, K-means clustering, maze routing, graph kernels, a client/server travel reservation system, and Delaunay mesh refinement. This covers many application areas in which STMs have been used, but do not represent codes from the area of high performance computing, for which HTM is a promising approach to overcome synchronization overheads and to improve scaling of hybrid thread/MPI codes. In this paper, we therefore focus on a new benchmark explicitly designed to cover this area and present results that demonstrate how HTM can be deployed in HPC.

III. Experimental Setup

For all following experiments we use an early prototype of BG/Q installed at IBM. TM is available through IBM's XL C/C++/Fortran Compiler suite for BG/Q, which provides new language primitives that allow users to specify transactions.

A. Overview of BG/Q's TM Hardware

The BG/Q prototype we had access to had 32 nodes with 16 cores each. Each core can execute up to four hardware threads. Transactional memory is implemented within the L2 cache, which consists of 16 banks of 2 MB each located across a full crossbar from the 16 compute cores. The L2 cache has a cache line size of 128 Bytes. Memory accesses that can lead to conflicts between transactions, are tracked by the L2 cache, which is the point of coherency. Conflict detection between different transactions is completed in hardware, while conflict resolution is coordinated through the TM software stack. Note that, in addition to TM, the L2 cache also implements an improved set of atomic operations that targets faster thread synchronization. Comparisons in the remainder of the paper between TM and atomic operations therefore provide results between two novel and highly optimized schemes. More information on BG/Q's hardware in general can be found in a recent presentation by Haring at Hot-Chips [2].

B. Application Perspective in BG/Q's TM Software Stack

By default, the TM runtime uses a *lazy* (or optimistic) conflict detection scheme at commit time, as the runtime suppresses the hardware from sending interrupts to the conflicted threads as conflicts are detected. However, applications/users can enable a *pessimistic detection* scheme by setting the `TM_ENABLE_INTERRUPT_ON_CONFLICT` environment variable. That is, conflict arbitration happens immediately at the time of conflicts. Either scheme needs to be carefully chosen as an already doomed thread, if allowed to finish, may cause further spurious conflicts.

The TM runtime also relies on a lazy versioning (i.e., write-back) scheme as all speculative writes are buffered in the multi-versioned cache until commit time. Strong atomicity (i.e., opacity) is guaranteed unless a thread is running in irrevocable mode. In such a case, the thread runs non-speculatively and all writes take effect immediately. The `TM_MAX_NUM_ROLLBACK` environment variable controls when a thread should enter into irrevocable mode. The irrevocable mode is a mechanism that guarantees that a thread makes progress. The contention manager favors an older thread to commit based upon the timebase register value of the thread at the time when speculation starts. Aborting a transaction does not back-off for a pre-determined time, rather, a thread retries immediately. The runtime also implements flat nesting whereby commits and rollbacks are to the outermost TM region. As an additional feature, the runtime monitors the TM behavior of the application and provides the resulting TM statistics to the user. All TM statistics presented in this paper, are retrieved by this method.

C. The CLOMP-TM Benchmark

Since current TM benchmark suites do not provide the necessary coverage for scientific applications, we focus on the development of a new benchmark, specifically designed to expose the range of properties needed to characterize scientific workloads: CLOMP-TM[1]. It is designed to compare multiple synchronization constructs. In fact, it was created to help us mimic the application characteristics of several large scale, multi-physics applications used in production at DOE laboratories. It achieves this flexibility and wide coverage through a series of parameters that allow us to explore varying transaction granularities and conflict rates, coupled with typical computational kernels found in scientific codes.

CLOMP-TM originates from the publicly available CLOMP benchmark [24] used for evaluating OpenMP implementations. CLOMP resembles an unstructured mesh with a set of partitions. Each partition holds a linked list of zones. To vary the pressure on the memory system, the size of these zones can be configured. Computation, that is performed when updating a zone, only uses the first 32 bytes of each zone. The computation per zone update can be scaled by a factor. While the original CLOMP aims to quantify overheads due to threading and the specific OpenMP implementation, the CLOMP-TM aims at quantifying and comparing synchronization overheads of multiple synchronization constructs. CLOMP-TM can be configured to resemble the synchronization characteristics of typical scientific applications used in HPC. Thus, performance results of CLOMP-TM not only enable us to provide detailed characteristics of the low-level properties of the BG/Q, but also, and more importantly, to project the performance impact of TM on large scale parallel applications.

Major changes over CLOMP: In order to study the impact of TM in the presence of loop dependencies and the resulting conflicts, CLOMP-TM adds explicit and controllable dependencies to the loop structures of CLOMP. Besides the implementation with TM, CLOMP-TM also tests optimistic execution not secured by any synchronization[2] or using other constructs such as atomics or OpenMP-based constructs with the same level of abstraction in terms of programming.

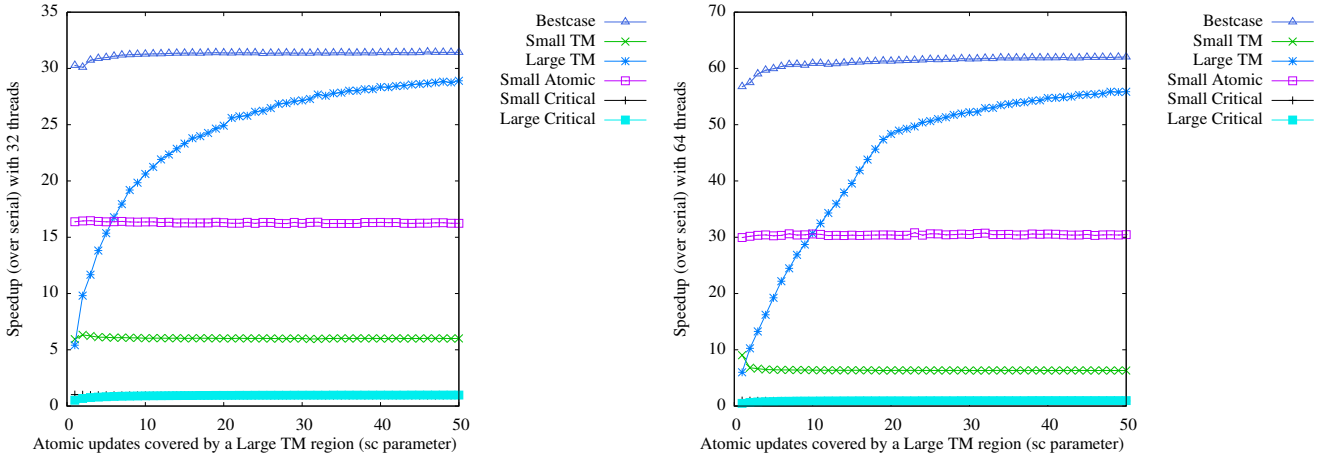
For a meaningful comparison of optimistic and pessimistic synchronization constructs, multiple memory access patterns have to be considered. These memory access patterns determine the likelihood of a conflict between concurrent accesses of two threads. A single parameter defines the zones that are updated by a thread. The contention arises when multiple threads update the same zones. These different contention scenarios are shown in Table I.

[1]For the experiments CLOMP-TM version 1.59 is used and will be made publicly available at publication time of the paper.

[2]In this configuration CLOMP-TM does not return correct results, but the timings can be used to study conflict free cases.

Name	Description	Contention
None	Threads do not conflict.	No contention.
Adjacent	Updates of adjacent memory addresses.	No to small contention.
Random	Repeatable but randomly seeming updates.	High contention.
FirstParts	Only the first parts are updated.	Highest contention.

TABLE I
DIFFERENT CONTENTION LEVELS IN THE CLOMP-TM BENCHMARK.



(a) 32 threads.

(b) 64 threads.

Fig. 1. CLOMP-TM performing 8 divide operations with a stride of 4 per zone update with excellent speedups of *Large TM* over *Small Atomic*. Run with `clomp-tm-bgq-divide4 -1 1 256 128 256 stride1,1,stride1%/2 sc 1 0 6 100`.

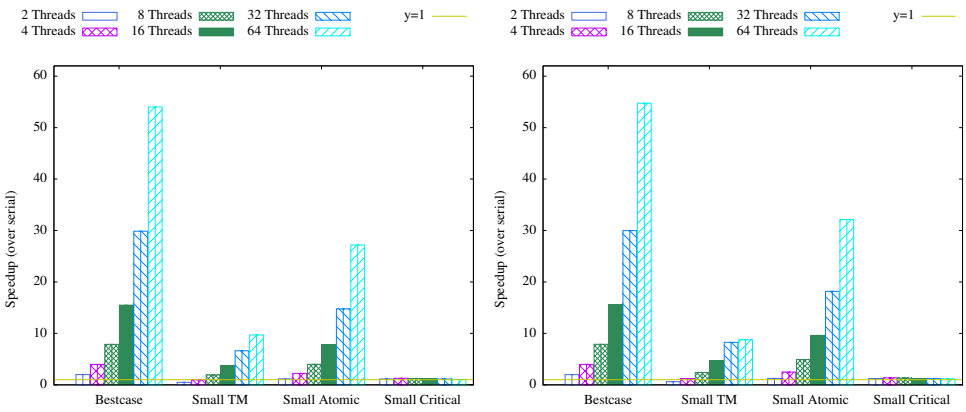
Name	Implementation	Description
Bestcase	—	Bestcase without synchronization.
Serial Ref	—	Serial reference implementation.
Small TM	`#pragma tm_atomic`	Synchronizing each update with a transaction.
Small Atomic	`#pragma omp atomic`	Synchronizing each update with an atomic operation.
Small Critical	`#pragma omp critical`	Synchronizing each update with OpenMP's critical section.
Large TM	`#pragma tm_atomic`	All scatter zone updates in one transaction.
Large Critical	`#pragma omp critical`	All scatter zone updates in one critical section.
Huge TM X	`#pragma tm_atomic`	X times *Large TM* in one transaction.

TABLE II
DESCRIPTION OF SYNCHRONIZATION CONSTRUCTS USED IN CLOMP-TM.

In comparison to the CLOMP benchmark, the updates of a zone are enlarged. This new construct is called *scatter zone* and enables larger critical sections, which resembles the update of multiple variables (e.g., coordinates with multiple dimensions x, y, and z) in one critical section. For the *large* versions of the synchronization constructs, the parameter *scatterCount* defines the number of updated zones in a single synchronized block.

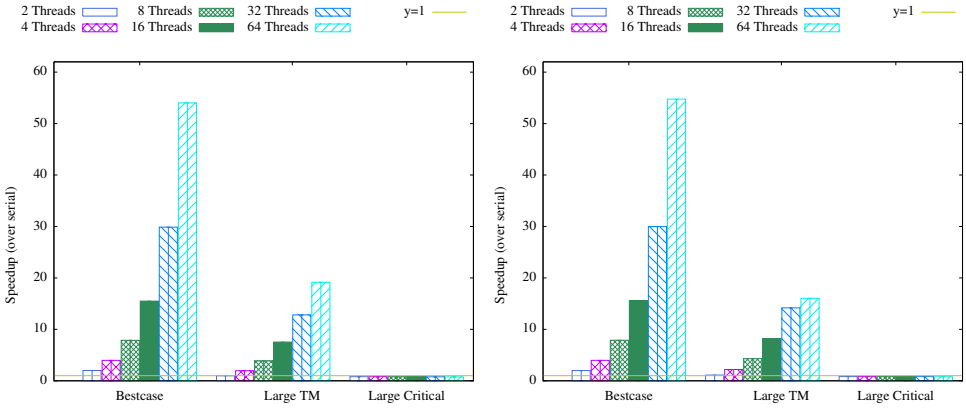
Each iteration executes the selected computation pattern. Available patterns with increasing complexity are: *none*, *divide*, *manydivide*, and *complex*. *None* performs no computation, *divide* performs a single, *manydivide* multiple, floating point divide operations and *complex* exercises math functions e.g., `log` and `sqrt`. CLOMP-TM is carefully designed to eliminate as much noise as possible: I/O is performed only outside of timing loops and all the loops are through with several iterations just before the timing loops to eliminate start up costs and cold cache effects. Table II holds the synchronization constructs to be compared.

Comparison of CLOMP-TM with TM benchmarks: Apart from the heavily cited STAMP benchmark suite [23] that does not represent the scientific application behavior we are interested in, a growing number of parameterized workloads gains popularity. An important example is the WormBench workload [25]. WormBench is derived from the popular snake game and has been designed to evaluate and verify the efficiency of a TM system. WormBench is written in *C#* and enables performance comparisons between TM and global locks. Parameters are size of the world (matrix), number of worms (threads), body and head size of a worm and operations to be performed while moving. In contrast, CLOMP-TM enables a comparison with a single instruction atomic update, an unsynchronized version and two sizes of transactions and critical sections, respectively. For our workloads *C#* does not play an important role and we believe that a more versatile benchmark to model scientific workloads is needed, which lead us to develop CLOMP-TM. A more suited candidate for the modeling of arbitrary TM workloads is Eigenbench [26]. Eigenbench uses orthogonal metrics to model a specific workload. For our use case, not knowing *a priori* how TM will perform, we would have to rewrite the application with transactions, measure the metrics, derive a configuration for Eigenbench and use this to model our workload. With CLOMP-TM, the user only needs to know the number of shared memory accesses and the number of floating point operations per loop iteration to set the right parameters that will resemble the application behavior.



(a) Speedup with *None*. (b) Speedup with *Random*.

Fig. 2. CLOMP-TM performing 8 divide operations per zone update with small critical sections.



(a) Speedup with *None*. (b) Speedup with *Random*.

Fig. 3. CLOMP-TM performing 8 divide operations per zone update with large critical sections.

IV. Characterizing TM Performance on BG/Q using CLOMP-TM

For the CLOMP-TM cases presented in this paper, synchronization overheads can dramatically affect speedup. We vary the parameters of CLOMP-TM to learn how the parameters affect the speedup and to find out what code properties qualify for TM. These results will help application developers to tune their codes (e.g., through picking a better suited synchronization primitive), but the achievable speedup is determined by the properties of the application (e.g., ratio of computation and synchronization, contention for memory locations).

For our initial experiments targeted at understanding the potential for TM, we chose the parameters for CLOMP-TM in a way that TM outperforms a highly efficient implementation of `omp atomic`. In this configuration, CLOMP-TM performs 8 divide operations per zone update with a stride of 4. Threads do not contend for memory locations. We increase the size of the scatter zone so that an increasing amount of updates are carried out in *Large TM*. Figure 1 a) illustrates that in the case of 32 threads performing 5 zone updates is the cross-over point for *Large TM* over *Small Atomic*. For 64 threads the number of zones is twice as high (see Figure 1 b)). Please note that the large amount of computation per zone update masks the overheads of synchronization.

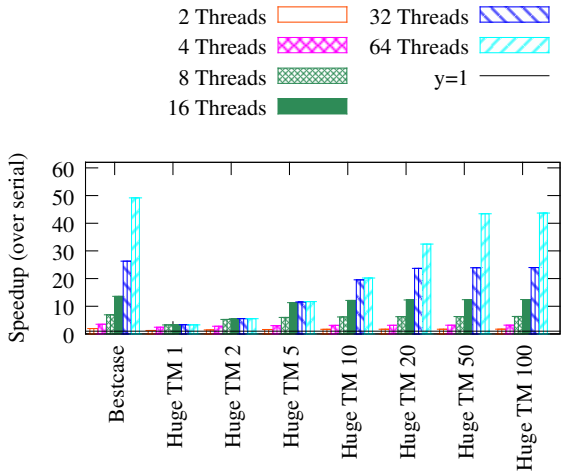
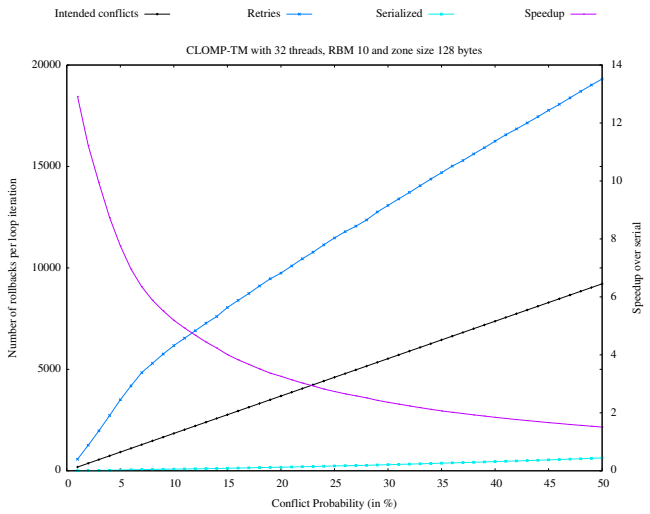
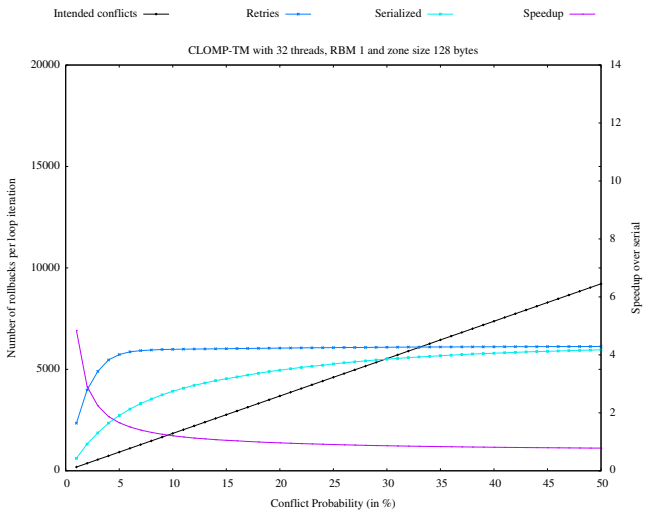


Fig. 4. CLOMP-TM (v. 1.36) performing 8 divide operations per zone update with huge critical sections. Speedup shown with *None*.



(a) 32 threads with RBM 10.



(b) 32 threads with RBM 1.

Fig. 5. Influence of `TM_MAX_NUM_ROLLBACK` (RBM) on retries/speedup. Run with `clomp-tm-bgq-divide1 -1 1 x1 d6144 128 firstZone,`*cp*`,randFirstZone 3 1 0 6 100`.

A. Synchronization Overhead

To understand the tradeoffs in using TM, we first study the synchronization overhead associated with different approaches and contrast them to the TM results. We obtain the results in this section by using the parameters shown in Table III. Memory is allocated by the main thread. This is sufficient because memory accesses are uniform in BG/Q. The setting of *zonesPerPart* equal to 100 stems from the original CLOMP and mimics the loop sizes of many multiphysics applications [24]. The chosen computation pattern is *divide*. For each zone update 8 extra *divide* calculations are executed. The environment variable `OMP_WAIT_POLICY` has been set to `ACTIVE` for all runs.

Figure 2 shows the speedup of the different synchronization mechanisms for updating one memory location. *Small Atomic* achieves the highest speedup for a small critical section compared to *Small TM* and *Small Critical*. Also, with contention, generated by the *Random* and *firstParts* (not shown) memory access patterns, *Small Atomic* provides the fastest synchronization.

numParts	64
zoneSize	128
zone alignment	128
scatter	3
flopScale	1
timeScale	100
Zones per Part	100
Total Zones	6400
Zone Calc Stride	1
Extra Zone Calcs	8
Zone Calc Flag	-DDIVIDE_CALC
Zone Calc Formula	((1.0/(x+2.0))-0.5)

TABLE III
PARAMETERS FOR CLOMP-TM.

Large TM outperforms *Large Critical* as can be seen for both no and high contention cases of Figure 3. Thus, for critical sections with more than one memory update, TM is the preferred method. The *Huge TM* with 100 times the size of *Large TM* performs extremely well in case of no contention (see Figure 4). Further experiments with higher contention cases reveal that this speedup is very fragile. These experiments demonstrate (and the TM statistics confirm) that longer transactions are more susceptible to contention and should be deployed with great care.

B. Conflict Probability

The performance of any TM application depends on the number of conflicts it has to encounter that lead to potentially costly rollbacks. We study this issue by extending CLOMP-TM with a special mode that enables transition between scatter modes. Thus, a parameter has been added that defines the number of *intended conflicts* for a run. For our experiments we compute this parameter from a *conflict probability* (*cp*) as follow: $total\ zones * scatter * cp$. Updates are counted as *intended conflicts* and performed inside a large transaction. Note, however, that not all *intended conflicts* lead to an actual conflict and some conflicts can cause multiple rollbacks.

Figure 5 a) illustrates the impact of the number of retries on the achievable speedup. We can clearly see that a linear increase of the conflict probability (shown as intended conflicts) leads to an exponential decrease of the speedup. In this experiment the zone size is set to 128 bytes. In case it is smaller (e.g., 64 or 32 bytes) conflicts may be falsely detected because two zones are mapped to the same cache line. These *False Positives* are eliminated when the zone size equals the size of the L2 cache line.

C. Tuning the BG/Q TM Runtime Environment

The TM runtime system in BG/Q provides a series of parameters that can be used to fine tune the performance of HTM applications. These parameters are available to the user through environment variables that can be set before the code's execution. The most significant one is `TM_MAX_NUM_ROLLBACK` (RBM), which controls the number of times a transaction can be aborted and rolled back before the runtime gives up on it and executes it in *irrevocable mode*, i.e., the transaction is executed non-transactionally under a global lock so that other transactions can not interfere. The TM runtime will further mark this transaction and execute it in irrevocable mode right away on a subsequent execution.

Figure 5 shows results with 32 threads and RBM set to 1 and 10. In Figure 5 a) RBM is set to 10 and shows a significant higher number of retries than Figure 5 b) (RBM 1). The relative number of serialized transactions is higher for RBM 1. Both observations are due to the RBM setting, since a smaller RBM value serializes after fewer retries. In terms of speedup, RBM 10 outperforms RBM 1 due to less frequent serialization. The effects of the adaptive TM runtime on the performance need closer investigation.

Figure 6 demonstrates the influence of RBM 1, RBM 5 and RBM 10 on the achievable speedup with TM. For all contention levels and *Small TM* the differences between RBM 5 and RBM 10 are insignificant. For higher contention and *Large TM*, RBM 10 has slight advantages over RBM 5. RBM 1 shows the worst performance for the presented level of contention.

A second important parameter for TM is the scrub rate. It triggers a garbage collection for TM SpecIds, which mark entries in the cache as belonging to the same or different transactions. Figure 7 shows that varying the scrub rate has a large impact. For our benchmark with a lot of transactions and short intervals between these, a scrub rate of 6 is best.

D. CLOMP-TM with mixed Scatter Modes

So far, we have only discussed settings with a single scatter mode at a time (see Table I). This leads to a fixed TM application behavior that defines the contention between threads for the whole program run. As a result, TM either performs excellent because of the lack of conflicts (e.g., scatter mode None) or suffers from the frequent retries (e.g., firstParts). This model, while useful to get point results, is too restricted to model all scientific workloads and expose the potential of TM. CLOMP-TM therefore additionally supports two different scatter modes that execute alternately. It uses a parameter to define how often the second scatter mode will be used, i.e., increasing this parameter leads to more updates with the second scatter mode.

E. Using TM in the Context of MPI Applications

Up to this point, we focused on single node experiments using OpenMP as the method for threading. To work across nodes and hence to exploit the vast parallelism available in BG/Q systems, scientific applications will require additional parallelization with MPI. Consequently, it is important to

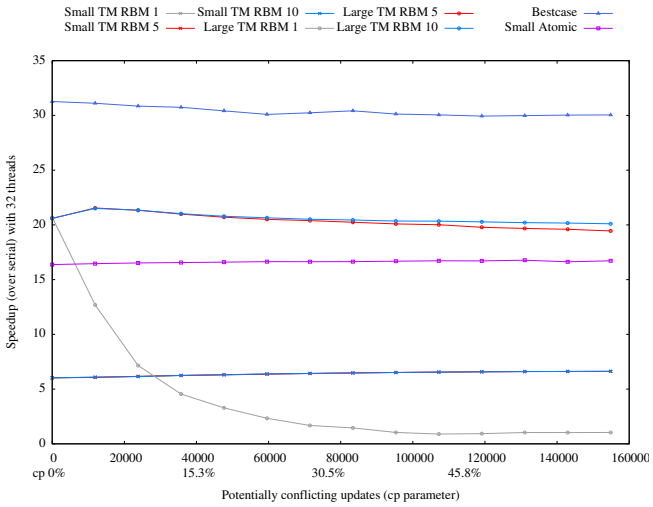


Fig. 6. Studying the influence of setting `TM_MAX_NUM_ROLLBACK` with CLOMP-TM and changing the level of contention. Run with `clomp-tm-bgq-divide4 32 1 256 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`.

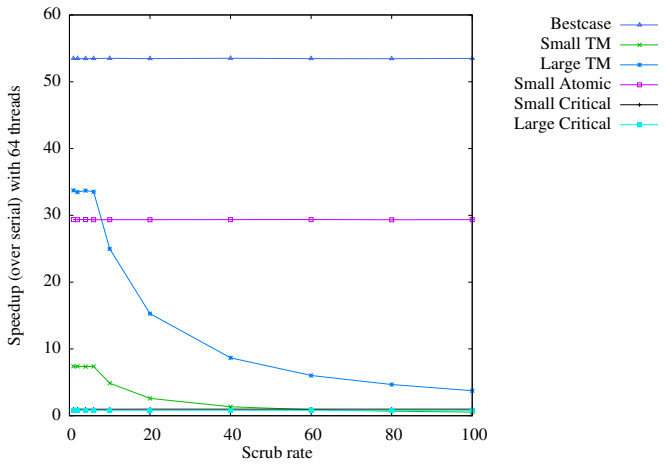


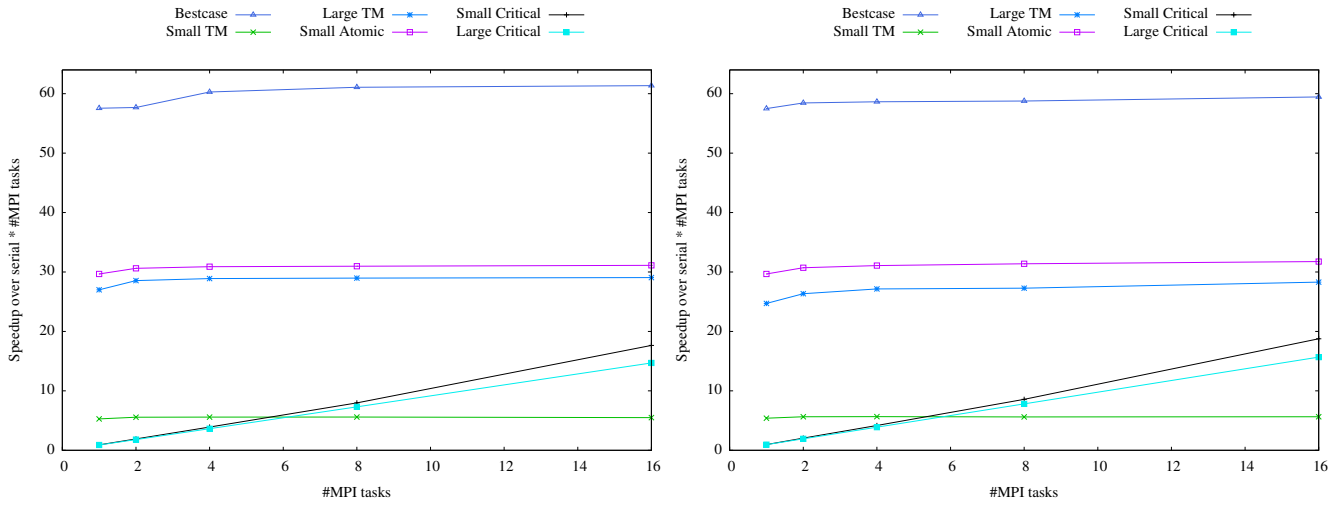
Fig. 7. Influence of the scrub rate for SpecIds on performance with 64 threads. Run with `clomp-tm-bgq-divide1 -1 1 64 100 128 InPart,10,firstParts 10 1 0 `*sr*` 100`

understand the interplay between OpenMP threading with TM support and having multiple MPI tasks on the node.

In the following we study the side effects of running multiple MPI tasks, each executing CLOMP-TM, on one node. Our goals are:

1) to verify the robustness of the results above,
2) identify bottlenecks due to the sharing of architectural resources,
3) and determine a fitting MPI task to OpenMP thread ratio.

We extended CLOMP-TM to execute multiple instances of its core functionality, synchronized by MPI operations. Besides calls to init and finalize MPI, we inserted *MPI_Barrier*s. These barriers are placed such that all MPI tasks execute the code for the same synchronization primitive. An example for the placement of the *MPI_Barrier* calls is shown in Listing 1. To execute in this lock step fashion guarantees that all MPI tasks execute the code for the same synchronization primitive. As



(a) No to low contention (rollback per transaction ≈ 0).

(b) High contention (rollback per transaction ≈ 1).

Fig. 8. CLOMP-TM with MPI performing 8 divide operations with a stride of 4 per zone update with no and high contention. Run with `clomp-tm-mpi-bgq-divide4 -1 1 (1024/taskno) 128 256 stride1,`*cp*`,stride1%/2 10 1 0 6 100`. *cp* is set to $\frac{16}{taskno}$ for the left and $\frac{1.12*10^6}{taskno}$ for the figure on the right.

```
MPI_Barrier(MPI_COMM_WORLD);
get_timestamp (&bestcase_start_ts);
do_bestcase_version();
get_timestamp (&bestcase_end_ts);
MPI_Barrier(MPI_COMM_WORLD);
```

Listing 1. Use of MPI barriers for CLOMP-TM with MPI.

a consequence, we can directly control the contention on the architectural resources that are necessary for synchronization (such as the L2 cache). Thus, the methodology provides a clear and controllable mechanism to study the architectural resources needed by individual synchronization primitives.

Figure 8 illustrates the performance of CLOMP-TM with MPI for small and large critical sections using *strong scaling*, i.e., the total amount of work is constant for all task counts. We achieve this by dividing the number of parts (initially 1024) as well as the number of updates in the second scatter mode by the number of MPI tasks. All MPI tasks execute as many threads as possible without oversubscribing the node (e.g., 1 MPI task executes 64 threads).

Figure 8(a) shows the average speedup of the threads in each MPI task multiplied by the number of MPI tasks on the y-axis. The number of MPI tasks is plotted on the x-axis. In this case with extremely low contention, *Large TM* performs almost as well as *Small Atomic*. The surprise is that for large task counts, *Small* and *Large Critical* perform better than *Small TM*. Especially *Small Critical*, which is protecting one memory location, has been optimized heavily in the new BG/Q L2 cache and is now a strong alternative for TM if the granularity in the codes allows for this. With increased contention (Figure 8(b)) *Large TM* and *Small TM* perform slightly worse but the overall trend with respect to the critical section remains. *Large Critical* benefits from the smaller thread numbers at higher task counts because the cost for serialization is reduced.

F. Finding a Competitive Task to Thread Ratio

The architecture of one BG/Q node features 16 compute cores each equipped with 4-way hyper-threading. As demonstrated in earlier papers [24], an OpenMP barrier has a higher overhead for higher thread counts. Thus, a hybrid parallelization with MPI and OpenMP may achieve higher performance than an OpenMP only implementation. In order to be able to compare results of OpenMP and hybrid parallelization, we use a simple metric. For the hybrid case, we multiply the reported OpenMP speedup with the number of MPI tasks. Figure 8 shows that the *Bestcase* across MPI tasks is stable. Across all tested memory access patterns and MPI tasks configurations, the OpenMP version with the highest possible thread count performs best. While this is not surprising since the BG/Q architecture requires at least two threads per core to be able to reach full instruction issue bandwidth, it is nevertheless an important first insight that we gain from this experiment. For architectures with hyper-threading, the additional HW threads are often turned off because they lead to a slowdown. For the BG/Q architecture running the CLOMP-TM (with MPI) benchmark this is not the case. Every thread (even beyond the minimum of two needed for full issue bandwidth) contributes an important part of the reported performance. For the executed strong scaling experiments, however, the results of finding a preferable task to thread ratio are inconclusive. All tested ratio perform well and differences are extremely small.

In all high contention cases *Small Atomic* performs best. For cases with little to no contention *Large TM* may perform almost as well as *Small Atomic*. The large transactions benefit from the optimistic concurrency and the overhead for setting up the transaction is amortized due to the long transaction size. Unfortunately, this effect is limited to scenarios where expensive roll back operations are infrequent.

V. Lessons learned

The experiments described above give us a clear characterization of HTM on BG/Q and provide the necessary information to understand which kind of applications can benefit from HTM. In the following we summarize these findings in a set of best practice guidelines that will help code developers on BG/Q decide if and how to best exploit HTM.

In particular, codes that exhibit the following properties are likely candidates for HTM:

- critical section should have low contention so that conflicts are unlikely,
- critical sections should access more than one memory location (preferably in the range of 10 to 20) so that `omp atomic` is not applicable and TM's property of providing atomicity for updates of multiple memory locations is valuable,
- high computation to synchronization ratio so that computation can mask the overheads of synchronization.

For synchronization with OpenMP, both the size of the code region that needs to be executed atomically and the potential conflict rate play an important role:

- For code regions that only require atomic updates using one instruction, *omp atomic* shows the best performance, since it can be mapped to the efficient atomic instructions implemented in the BG/Q L2 cache.
- For larger critical sections with low to moderate contention and conflict potential ($<<$ 1 rollback per transaction), TM using the *tm_atomic* primitive is beneficial, since the costs of conflicting transactions are amortized by avoiding serialization.
- In case of very high contention ($>$ 1 rollback per transaction) and small critical sections, *omp critical* also outperforms TM, since TM conflicts and rollbacks start dominating leading to higher overhead.
- For applications that are not utilizing the full memory bandwidth with a high transactional execution time and short times in between transactions, setting the scrub rate to 6 yields better performance.

These findings complement a previous study on using Software Transactional Memory for scientific codes using a different and more specific setup [27]. Additionally, researchers already identified codes that match the criteria from above and are expected to benefit from TM [28]. This work was limited to STM methods and has only recently been verified on a HTM system, publishing first performance results of the BG/Q architecture [3]. Our current recommendations verify the applicability of these previous preliminary studies to HTM, extend them by adding tradeoffs offered by the new adjustable performance parameters found in IBM's HTM solution, and generalize them to a more comprehensive guide for application developers.

VI. Application Case Studies

A. MCB: A Proxy Application for Monte Carlo Simulations

In this section we apply the best practices from the previous section to a benchmark closely representing a real world application. The Monte Carlo Benchmark (MCB) models a Monte Carlo simulation, a popular technique for physics simulations. In contrast to classical simulation approaches, Monte Carlo simulations do not compute their result explicitly, but instead adaptively sample the simulation domain and execute individual simulations for each sample. This process is repeated until the probability of a result can be quantified.

The initial MCB code was already parallelized with MPI and OpenMP using *omp critical* and *omp atomic* to synchronize OpenMP threads (denoted as *Critical & Atomic* in the following). As a first, naive TM implementation, referred to as *TM naive*, we replace all critical sections with transactions and set the TM environment variables to their default for *TM simple* and *Critical & Atomic*.

Additionally, we create an optimized version, called *TM opt*, following the lessons in the previous section. In *TM opt* we use a hybrid strategy matching the characteristics for each synchronization construct: synchronizations that involve only one instruction use *omp atomic*, while all *omp critical* constructs are replaced with *tm_atomic*.

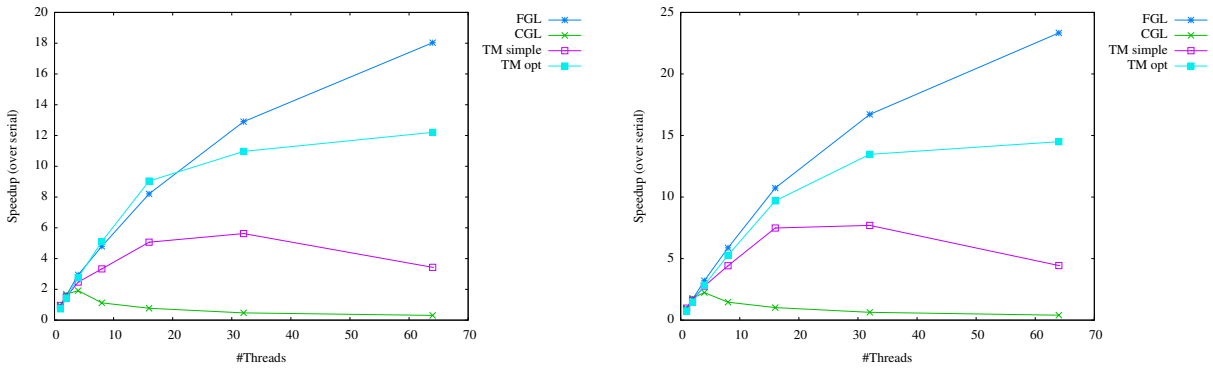
Table IV shows the results of the experiments with one MPI task and 64 threads in a strong scaling experiment with $5*10^6$ particles. Each value is an average of "samples per second" over 30 runs and normalized to baseline: "samples per second" with one MPI task and one thread. *TM opt* has a speedup over baseline of 27.45 and performs almost as good as the original version, but at reduced code complexity and programmer effort. The result of *TM naive* demonstrates that the lessons learned in this paper are essential to getting good performance. Further experiments reveal a limited potential for optimizing the synchronization of threads in MCB. Commenting out all occurrences of `omp atomic` and `omp critical` (and ignoring the fact that this results in wrong answers for the simulation) yields $\approx 5\%$ performance improvement.

B. Fluidanimate from the PARSEC suite

In addition to the Monte Carlo Benchmark, we use *fluidanimate* from the PARSEC benchmark suite [5]. *fluidanimate* implements a Smoothed Particle Hydrodynamics (SPH) method to animate fluid dynamics. To include it in the PARSEC suite, the application has been parallelized with Pthreads and fine-grained locking. Under the assumption that particles can not travel more than one cell in one time step, this parallelization uses an array of locks to protect the boundaries. An if-statement checks whether a lock needs to be taken.

Code version	*Critical & Atomic*	*TM naive*	*TM opt*
Speedup	27.57	20.06	27.45

TABLE IV
MCB WITH ONE MPI TASK AND 64 THREADS (STRONGSCALING) – SPEEDUP OVER BASELINE.



(a) Input data set *simlarge*.

(b) Input data set *native*.

Fig. 9. *fluidanimate* with coarse-grained and fine-grained locking as well as simple and optimized transactions.

This synchronization pattern is rather sophisticated and by far exceeds the complexity of a single global lock. Because the programming complexity of TM can be compared with a single global lock, we added two versions: one with a coarse grain lock (cgl) and a simple TM version (TM simple) that replaced all lock acquire and lock release operations (that occur in three code segments) with a transaction.

Then, we apply the lessons learned from Section V. First, we enlarge all three transactions by removing the if-statement and changing the two outer loops to be inner loops. More measurements reveal that for the third transaction having three nested inner loops is best. Further, we reduce the scrub rate to 6 so that SpecIds from the TM hardware will be reclaimed faster. All measures combined deliver the performance shown for *TM opt* in Figure 9. For the small input data set (simlarge) and medium thread counts, *TM opt* outperforms the fine-grained locking (cf. Figure 9 a)). For the larger input data set (native) the lessons learned are necessary to increase the scalability with TM and come into sight of fine-grained locking (Figure 9 b)). The execution with native input data sets increases the input data size and the frame rate simultaneously. We were interested to know which of these parameters influences the performance in favor of TM. The result is that the smaller input data favors TM whereas the frame rate simply serves as a multiplier of the observed performance. Further, this experiment reveals that the BG/Q architecture is extremely stable with very little noise. The observed performance of the synchronization patterns shows that even with Hardware Transactional Memory, expert-level use of lightweight efficient fine-grained locks will be hard to beat. Moreover, we identified the need to research tools with support for TM that enable an in-depth understanding of the TM behavior and the causes for performance degradation. From a programmability perspective, employing TM is as simple as using a single global lock. For this experiment, even the unoptimized TM version outperforms the single global lock in terms of speedup and scalability. Therefore TM takes an important step towards simplifying shared memory programming.

VII. Conclusions

In this paper we evaluated BG/Q's TM hardware from the perspective of an application developer. We introduced CLOMP-TM, a benchmark designed to represent scientific applications, and used it to contrast HTM against traditional synchronization primitives, such as *omp atomic* and *omp critical*. We then extended CLOMP-TM with MPI to mimic hybrid MPI/OpenMP parallelization. Additionally, we studied the impact of environment variables on the performance. Finally, we condensed the findings into a set of best practices and applied them to a Monte Carlo Benchmark. An optimized TM version of MCB with 64 threads achieved a speedup of 27.45 over the baseline. Further, an optimized TM version of the Smoothed Particle Hydrodynamics method from the PARSEC suite achieved a speedup of 14.5 with 64 threads and significantly outperformed a simple TM version (speedup of 4.4) as well as a coarse grain lock (speedup below 1) and verified the usefulness of the best practices. Moreover, our results also show that an expert-level use of lightweight efficient fine-grained locks is hard to beat with TM. For MCB, a synchronization pattern that combines omp atomic and omp critical achieved a slightly larger speedup of 27.57 over baseline. These findings illustrate that performance with HTM does not come for free and, even when following the guidelines developed and presented in this paper, performance with TM may not quite measure up to the expert-level use of locks for the scientific applications considered. However, TM comes with the advantage of improving the programmability and productivity because the user does not have to explicitly manage locks (which is known to be error-prone). Thus, the use of TM or locks depends on the expected gain when comparing development effort with performance improvements. Our findings motivate further research to refine our lessons learned and develop tools for programming with TM.

Acknowledgment

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-556571).

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. New York, NY, USA: ACM, 1993, pp. 289–300. [Online]. Available: http://doi.acm.org/10.1145/165123.165164

[2] R. Haring, "The Blue Gene/Q Compute Chip," in *Hot Chips 23*, August 2011. [Online]. Available: http://www.hotchips.org/archives/hc23/HC23-papers/HC23.18.1-manycore/HC23.18.121.BlueGene-IBM_BQC_HC23_20110818.pdf

[3] B. L. Bihari, M. Wong, A. Wang, B. R. Supinski, and W. Chen, "A Case for Including Transactions in OpenMP II: Hardware Transactional Memory," in *OpenMP in a Heterogeneous World*, ser. Lecture Notes in Computer Science, vol. 7312. Springer Berlin Heidelberg, 2012, pp. 44–58. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30961-8_4

[4] A. Wang, M. Gaudet, P. Wu, M. Ohmacht, J. N. Amaral, C. Barton, R. Silvera, and M. M. Michael, "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," in *International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*, 2012, Accepted for publication.

[5] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[6] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*. Morgan & Claypool Publishers, 2010, vol. 5, 2nd edition, Synthesis Lectures on Computer Architecture.

[7] P. Felber, C. Fetzer, and T. Riegel, "Dynamic Performance Tuning of Word-Based Software Transactional Memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. New York, NY, USA: ACM, 2008, pp. 237–246. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345241

[8] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, 2006, pp. 194–208.

[9] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, "Anatomy of a Scalable Software Transactional Memory," in *TRANSACT'09: Workshop on Transactional Computing*, February 2009.

[10] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2006, pp. 187–197.

[11] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software Transactional Memory: Why is it Only a Research Toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[12] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A Chip-Multiprocessor with Transactional Memory Support," in *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, april 2007, pp. 1 –6.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Jun 2004, p. 102.

[14] C. Kachris and C. Kulkarni, "Configurable Transactional Memory," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 65–72. [Online]. Available: http://dl.acm.org/citation.cfm?id=1302498.1303053

[15] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural Support for Software Transactional Memory," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 185–196.

[16] Y. Lev, M. Moir, and D. Nussbaum, "PhTM: Phased Transactional Memory," in *2nd Workshop on Transactional Computing (TRANSACT '07)*, August 2007.

[17] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid Transactional Memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. New York, NY, USA: ACM, 2006, pp. 336–346. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168900

[18] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of AMD's Advanced Synchronization Facility within a Complete Transactional Memory Stack," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. New York, NY, USA: ACM, 2010, pp. 27–40. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755918

[19] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early Experience with a Commercial Hardware Transactional Memory Implementation," *SIGPLAN Not.*, vol. 44, pp. 157–168, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1508284.1508263

[20] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *Micro, IEEE*, vol. 29, no. 2, pp. 6 –16, march-april 2009.

[21] S. Kang and D. A. Bader, "An Efficient Transactional Memory Algorithm for Computing Minimum Spanning Forest of Sparse Graphs," *SIGPLAN Not.*, vol. 44, no. 4, pp. 15–24, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504182

[22] C. Click, "Azul's Experiences with Hardware Transactional Memory," Jan 2009, in HP Labs - Bay Area Workshop on Transactional Memory.

[23] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[24] G. Bronevetsky, J. Gyllenhaal, and B. R. De Supinski, "CLOMP: Accurately Characterizing OpenMP Application Overheads," in *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism (IWOMP '08)*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 13–25. [Online]. Available: http://dl.acm.org/citation.cfm?id=1789826.1789829

[25] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris, "WormBench: A Configurable Workload for Evaluating Transactional Memory Systems," in *Proceedings of the 9th Workshop on Memory Performance: Dealing with Applications, Systems and Architecture (MEDEA '08)*. New York, NY, USA: ACM, 2008, pp. 61–68. [Online]. Available: http://doi.acm.org/10.1145/1509084.1509093

[26] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A Simple Exploration Tool for Orthogonal TM Characteristics," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2010.5648812

[27] B. L. Bihari, "Applicability of Transactional Memory to Modern Codes," in *International Conference on Numerical Analysis and Applied Mathematics 2010 (ICNAAM 2010) Conference Proceedings*. Rodos, Greece: APS, 2010, pp. 1764–1767.

[28] M. Wong, B. L. Bihari, B. R. de Supinski, P. Wu, M. M. amd Y. Liu, and W. Chen, "A Case for Including Transactions in OpenMP," in *IWOMP 2010 Conference Proceedings*. Tsukuba, Japan: LNCS 6132, June 2010, pp. 149–160.